SootKeeper: Runtime Reusability for Modular Static Analysis

Florian Kübler[†] Patrick Müller[†] Ben Hermann*

Technische Universität Darmstadt, Germany

†{florian_vincent.kuebler, patrick.mueller}@stud.tu-darmstadt.de *hermann@cs.tu-darmstadt.de

Abstract

In order to achieve a higher reusability and testability, static analyses are increasingly being build as modular pipelines of analysis components. However, to build, debug, test, and evaluate these components the complete pipeline has to be executed every time. This process recomputes intermediate results which have already been computed in a previous run but are lost because the preceding process ended and removed them from memory. We propose to leverage runtime reusability for static analysis pipelines and introduce *SootKeeper*, a framework to modularize static analyses into OSGi (Open Service Gateway initiative) bundles, which takes care of the automatic caching of intermediate results. Little to no change to the original analysis is necessary to use *SootKeeper* while speeding up the execution of code-build-debug cycles or evaluation pipelines significantly.

CCS Concepts • **Software and its engineering** → **Reusability**; • **Theory of computation** → *Program analysis*

Keywords static analysis, modularity

1. Introduction

The development of static program analyses has become a complex job. Projects such as FlowDroid [2] (29,890 SLOC) or FlowTwist [9] (27,929 SLOC) already provide a large codebase and are themselves based on the frameworks Heros [3] (8,243 SLOC) and Soot [18] (379,248 SLOC). Ensuring a correct yet extensible codebase is not only a task for framework providers but also for developers of static analyses.

Therefore, components of these analyses have to be reusable and testable. In previous work, we advocated static modularity to improve the reusability and testability of analysis code [8]. Analysis concerns were split into separate independent classes so that they can be reused by configuration in order to express different analyses in the same domain (i.e. data-flow analysis). They can be tested both individually and in an integrated state.

However, even a well-designed and modularized analysis is always performed from scratch in every execution. Static analysis frameworks, such as Soot, perform a number of basic transformations (e.g. SSA) and analyses (e.g. a call graph analysis) on the code before the actual analysis under development is run. These basic steps may take up a considerable portion of the total runtime of an analysis, when considering the investigation of large code bases such as the Java Class Library (JCL). This can be problematic during the development of static analyses when rerunning the basic steps for each implement-test-debug cycle dramatically slows down development. It also can affect the overall runtime of an evaluation pipeline which compares different analysis implementations against each other.

In this paper, we propose to modularize static program analyses w.r.t. runtime reusability. We implemented our approach in a framework named *SootKeeper* which uses OSGi (Open Service Gateway initiative) modularity [12] to split analyses into small compartments which can be run separately and even in parallel. We leverage a key feature of OSGi modularity which is that modules (i.e. bundles in OSGi terminology) remain active in memory after a completed run. This allows us to speed up the debug cycle of downstream analyses by running only modified analysis modules using the precomputed results of the upstream analyses, which remain in memory as long as the framework is active or until they are invalidated. *SootKeeper* provides an API and infrastructure to develop, run, test and debug modular static analyses. Our approach is presented in Section 2.

By using code bases and analyses of various size and complexity we measured the possible speedup that can be achieved. Results show that our approach can significantly help in decreasing developer idle time during the development of static analyses when there is a stable overhead from upstream analyses. We present the results of these experiments in Section 3.

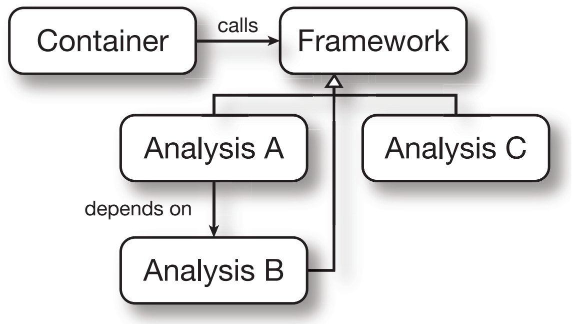


Figure 1: Example for an Analysis Pipeline in *SootKeeper*

We present related work and alternatives to our approach in Section 4 and conclude the paper in Section 5 with a summary and a look at future challenges.

2. Approach

Most implementations of static analyses can be seen as a pipeline of distinct modules, where there is a clear chain of dependencies. For instance, most analysis frameworks parse the targeted code and generate an intermediate representation in order to enable further analyses (e.g., control flow analysis, call graph computation, aliasing analysis). Results of such analyses themselves can be required for even more complex ones.

Splitting an analysis into modules can provide significant gains in term of maintainability, extensibility and reusability. With *SootKeeper* we aim to facilitate this while simultaneously providing runtime efficiency improvements. Developers of static analyses are able to reuse intermediate results from upstream analysis modules while working on downstream analysis modules or comparing different downstream analysis implementations in an evaluation pipeline.

SootKeeper is based on the OSGi modularity framework. Its components and the individual analysis modules are represented as OSGi bundles. *SootKeeper* consists of two main bundles: (1) the `framework` component, which provides the API that has to be used by every analysis module, and (2) the `container` component, which provides infrastructure to control analysis modules registered in *SootKeeper*.

Figure 1 shows an example scenario with three analysis modules. *SootKeeper*'s `container` and `framework` bundles are loaded (and active) within an OSGi container. The same holds for the bundles for analysis modules A, B, and C. Each analysis extends the `framework` bundle. In this example module A requires the results of module B for its analysis. The `container` controls the execution of the respective modules in the proper order and allows for interaction through OSGi actions (e.g. using the shell).

Transforming an Analysis After identifying modules in the analysis pipeline, all new analysis modules have to implement interfaces from the `framework` module in order to be applicable for *SootKeeper*. The `IAnalysisConfig` and `IAnalysisResult` interfaces are used to implement representations of configurations and analysis results. In order to bind an actual analysis to the module the `AbstractAnalysisService` class has to be extended. It is parameterized over the `IAnalysisConfig` and `IAnalysisResult` implementations of the module. Implementing classes have to provide the following information:

A name for the the analysis module by implementing the `getName` method.

Dependency information by implementing the `getDependOnAnalyses` method.

A configuration parser to leverage command line arguments for the current analysis in an implementation of the `parseConfig` method.

A configuration converter in an implementation of the `convertConfig` method which will be used to transform configuration options for upstream analyses the current modules depends upon.

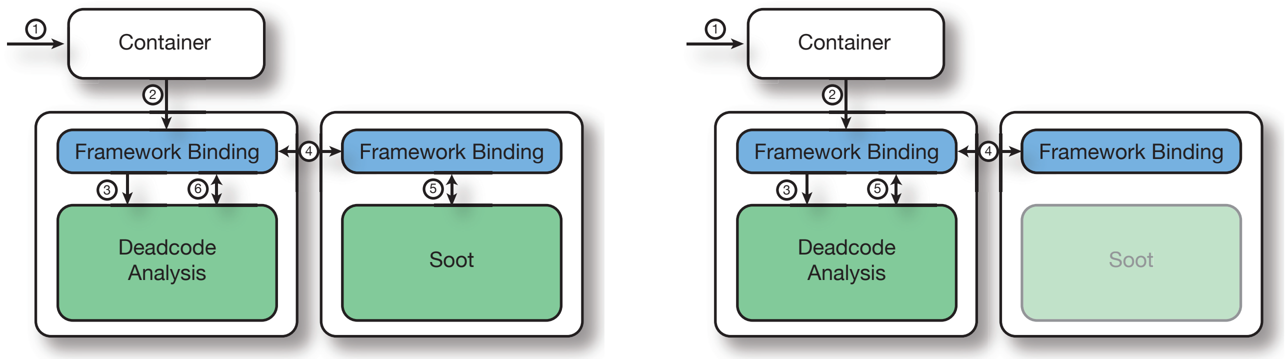
An analysis implementation in the `runAnalysis` method.

Finally, an analysis module has to provide an implementation of the `AbstractAnalysisActivator` class which handles the service registration in the OSGi lifecycle.

When running an analysis module the `framework` module takes care of several actions. All modules which this module depends on will be automatically executed in parallel (if their dependencies allow for this) using the correct configuration. The results of every individual analysis for each configuration will be cached for later use. If an analysis gets updated the corresponding caches will be cleared transitively, which also includes the results of analyses that depend on the updated one.

The Lifecycle of an Analysis As an example consider a dead-code analysis (DCA) which depends on preliminary analyses from Soot, e.g. control-flow graphs and an intermediate representation of the target program. Figure 2 provides an overview about the abstract control flow of *SootKeeper* in this example. In the following we explain the steps taken for the first run and all subsequent runs.

Figure 2a shows the execution of the dead-code analysis starting with the invocation of the container component (1). The container component identifies the bundle for the dead-code analysis and requests parsing of configuration data and the setup of the analysis component (3). During this process dependencies of the dead-code analysis module are resolved. Since there is no data from prior executions in the container component, the dead-code analysis module converts the configuration data to the format necessary for the Soot module and triggers its execution (4). As the Soot module does not have any additional dependencies, it configures and executes the requested analysis and stores the result in the container



a: First Run

b: Following Runs

Figure 2: Example of an Analysis Lifecycle

component (5). The result of this analysis is then used in the execution of the dead-code analysis (6) which in turn stores its result in the container component.

Now consider that the dead-code analysis is modified by its developer. She recompiles it and updates the module in the container. This will automatically invalidate the cached result of the previous DCA run. Nevertheless, Soot's result will be still in memory. Figure 2b shows the steps of *Soot-Keeper* taken in the next execution of the DCA module. Steps (1)-(3) remain the same in this process. However, during step (4) the existing result of the previous Soot run is found in the container. As the result has not been invalidated it is used in the DCA analysis in step (5) which stores its new result in the container for further downstream analyses (not shown here).

There is no retention policy for the caches, since the benefits of using *SootKeeper* are subverted, if upstream analyses are rerun very often with different settings. Caches are only cleared if the corresponding analysis or other upstream analyses are reloaded.

Integration For convenience, *SootKeeper* provides a Soot module which wraps Soot and all its non-OSGi dependencies in an OSGi bundle. This bundle implements the necessary *SootKeeper* interfaces and provides basic configuration and result classes. These may already be sufficient for other analyses without modification. For ease of configuration we use SootConfig[1], a fluent interface to configure Soot.

SootKeeper mainly targets Soot as an analysis framework. However, it is possible to use different Java-based frameworks such as OPAL [5] or WALA[2] by implementing a module in the same manner as our provided Soot module.

The built-in support for OSGi in common Java IDEs, like Eclipse and Intellij IDEA, makes a convenient debug process possible. Analysis modules in *SootKeeper* (and the platform itself) can be easily started and debugged with IDEs. While doing this, an analysis can be modified and updated, without losing the previous results of unchanged upstream analyses.

3. Evaluation

In order to evaluate *SootKeeper*, we inspect the effort necessary to convert existing analyses into analysis modules and the speed-up achieved during an average developer's day. This leads us to the following research questions:

RQ1 What is the effort of adapting existing analyses of various complexity to the requirements of *SootKeeper*?

RQ2 What is the average speed-up that can be expected by following *SootKeeper*'s approach?

Setup and Analysis Adaption We evaluated *SootKeeper* on four distinct analyses, which are all based on Soot. They are: (1) A simple intra-procedural dead code analysis [11], (2) an analysis finding direct usage of native code, (3) an analysis of intrusive usage of reflection and (4) FlowDroid [2]. While the first three are executed against a set of 66 Java common open-source libraries, we executed FlowDroid against a set of 21 Android applications. We selected the analysis targets in order to provide various complexities and code base sizes for the analyses.

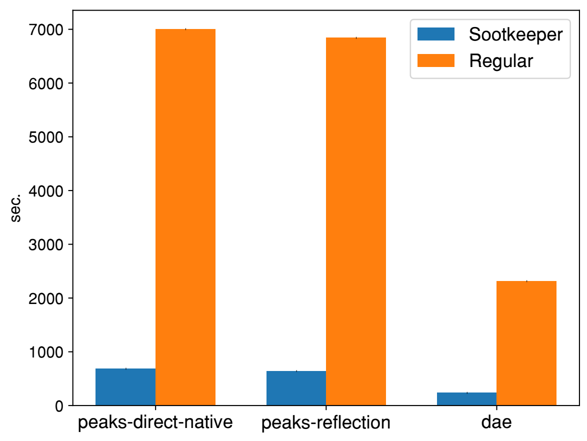
All analyses, except the dead code analysis, existed beforehand and needed to be modified to run in *SootKeeper*.

We implemented the dead code analysis ourselves as a classical Soot transformer. Due to internal requirements of Soot, it was necessary to add this transformer to a pack in Soot before performing the analysis and to remove it afterwards. This adaption took about an hour of work for one developer. We added 7 and changed 23 lines of code from the original analysis.

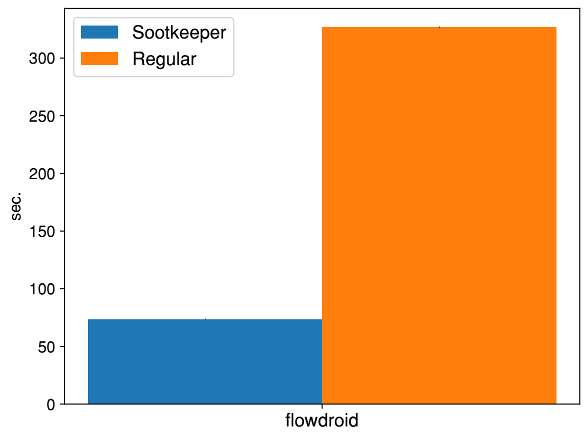
Both the direct native and the reflection analysis were part of the same project, which as a whole took less then 30 minutes of work to be modified. All analyses in that project were already structured to be able to run after Soot, i.e. they are not added to specific phases, therefore it was very easy

[1] `https://github.com/stg-tud/sootconfig`

[2] `http://wala.sourceforge.net/wiki/index.php/Main_Page`



a: Analysis Runtime for rt.jar



b: Analysis Runtime for the Ebay Kleinanzeigen Android Application

Figure 3: Excerpts from Experiment Results

to convert them to use *SootKeeper*. We added 715 lines of code. No other changes were necessary.

FlowDroid's conversion was the most complex one, since this project just uses Soot's infrastructure and runs the preliminary analyses it needs from Soot itself. Therefore, we have split FlowDroid into two smaller analyses: (1) A base analysis which handles the main method generation and source/sink lookup and (2) the actual taint tracking. Details of the different steps can be found in the work of Arzt et al. [2]. Only minor modifications of FlowDroid itself were necessary, i.e. visibility changes of certain attributes. In total 168 lines where changed in FlowDroid itself, with most of the changes in extra maven files. Most of the adaption took place in additional classes implementing or extending FlowDroid in addition to those necessary for *SootKeeper*. This additional code contains 1274 lines of code including two maven `pom.xml` files. One of the authors took approximately 8 hours to convert FlowDroid, since its internal structure is complex and it was necessary to find an optimal position for the split. However, included in these 8 hours is a conversion of the complete FlowDroid project to the maven structure in order to enable easier builds of OSGi bundles.

In summary for RQ1, we found that the effort needed to adapt an analysis varies over the complexity of its design. As being a rather complex analysis, FlowDroid can be seen as an upper-bound scenario for the adaption to *SootKeeper*.

To answer RQ2, we performed executions of the aforementioned analysis on various targets. In order to model a developer's behavior, we assume that a developer of program analyses has an average of ten implement-test-debug-cycles per day. Therefore, we measured the execution times of 10 runs of each analysis for each approach (*SootKeeper* and the traditional approach). For the *SootKeeper* approach we updated the analysis after each run without rebuilding the analysis, i.e. its caches were cleared, but not the caches of the analyses it depends on.

All experiments were conducted on a machine running OS X 10.12 with a 8-core Intel Xeon E5 3.0 GHz processor and 32 GB memory inside a Docker container [10]. As Java Runtime Environment, we used the OpenJDK 1.8.0_111 release, with a heap size set to a maximum of 16 GB.

In order to minimize the hardware requirements, FlowDroid was configured using the recommended settings for speed[3].

Results We measured an average speedup of 7.95 for the Java-targeting analyses and 4.58 for FlowDroid. As an example Figure 3a shows the results for the three different analyses on the rt.jar[4]. Figure 3b shows the results for the FlowDroid analysis of the ebay Kleinanzeigen Android application.

In contrast to the Java analyses, FlowDroid does not benefit from as large a speedup, since its partitioning is not as asymmetrical as in the other cases. The taint tracking part, may easily take more execution time than the earlier parts, depending on the current settings and the analysis target. However, we were always able to achieve at least a minimal speed-up (of approximately 2 times) and added only neglectable overhead.

In order to facilitate the reproducibility of these results we have created docker images. There is a base image[5] which contains a prebuilt version of *SootKeeper* and the Apache Felix OSGi distribution[6]. We additionally supply another

[3] `https://github.com/secure-software-engineering/soot-infoflow-android/wiki`

[4] Oracle JDK8 Update 121

[5] `https://hub.docker.com/r/patrickmueller/sootkeeper/`

[6] `https://felix.apache.org/`

image[7], which provides prebuilt versions of the analyses we ran as part of our experiments. Detailed instructions on the usage of this image and a list of our analysis targets can be found in the *SootKeeper* repository on GitHub[8].

4. Related Work

Work related to *SootKeeper* can be split into work on speeding up analyses by reusing information and work on modularization of program analyses.

The idea of avoiding redundant resource expensive computations in program analyses is not a new one. Visser et al. propose Green [19], a SAT solver interface for constraints used by symbolic execution. They check if results have been already computed and make them persistent across different runs, analysis targets, analyses and physical locations. Beside the performance improvements they refactored an existing dynamic symbolic execution analysis which also made use of Soot. Another approach is to store intermediate analysis results as metadata. As an example Probst [13] stores constraint graphs of libraries used in callgraph construction. Ramírez-Deantes et al. [14] use results of null, sign or heap analyses as input. Rountev et al. [16] compute and store IDE summaries of libraries.

However, storing computed information on the hard drive may entail multiple problems. Making complex data structures of the Soot framework persistent would require the refactoring of a large legacy code base. The data structures would have to be optimized for serialization regarding file sizes or input/output-speed. Also, this optimization would be specific to the data structures used and cannot be applied generally. With *SootKeeper* we achieve the same effect without touching Soot's source code at all. Furthermore, the stored data for big codebases (such as the JCL) can be very large. In many cases, the time taken for I/O will exceed the time needed to recompute the result and will be detrimental to the overall performance of the analysis. As *SootKeeper* retains the results of analysis modules in memory the same way as they have been computed there is little to no overhead over a non-modularized analysis.

SootKeeper is designed to increase the performance when designing, testing and debugging new analyses by caching results of required and unchanged analysis modules. A related scenario is the idea of using previous results to avoid re-computations of unchanged code, when analyzing a modified version of a target program. This approach is known as incremental analyses. Arzt and Bodden [1] present an approach for incremental analyses using the IDE frameworks. The work on incremental program analysis is based on Ryder [17].

To the best of our knowledge, there is no framework for the modularization of arbitrary program analysis frameworks and analyses. Nevertheless, there exists work on analyses and analysis frameworks that provide modularity to enhance extensibility and maintainability. Robby et al. [15] developed an extensible model checking framework having a plug-in based module system. The base framework is split into modules which are used to provide better maintainability and extendability. In prior work, we propose a design to separate IDE/IFDS flow functions [8] and thereby split analyses into different modules.

A different but related approach to our is to split the target program into modules and run analyses of the modules in parallel [4, 6, 7]. After the parallel run, the results are composed to a whole program analysis. This approach aims at performance gains through parallelization.

[7] `https://hub.docker.com/r/patrickmueller/sootkeeper-experiments/`

[8] `https://github.com/stg-tud/sootkeeper`

5. Conclusion

In this paper, we presented *SootKeeper*, a framework for analysis modularity which preserves intermediate results in an analysis pipeline. Analyses modularized in this fashion help to facilitate a more efficient development process and can, furthermore, help in streamlining an evaluation pipeline for that analyses. We observed that a speedup between a factor of 2 and 8 can be achieved for a regular development process and showed that this will increase with every further run. Additionally, the manual effort for the adaption of existing static analysis implementations to the framework remains in reasonable limits. For new implementations the effort is negligible in comparison to the implementation of the analysis itself. *SootKeeper* integrates very well into existing IDEs and build pipelines due to its adherence to OSGi. Developers can retain their workflow and all IDE features (including debugging) while still benefiting from *SootKeeper*'s infrastructure and speedup.

We provide the tool, its source code, two Docker containers, and the documentation at the following website:

`http://thewhitespace.de/projects/peaks/sootkeeper.html`

In future work we plan to extend *SootKeeper* to an integration platform for static analysis components to provide support for automated regression testing both in isolation and integrated with other components. This platform will also be the foundation for an automated evaluation environment for static analyses.

Acknowledgments

The authors would like to thank Ben Setzer for his initial implementation of *SootKeeper* and Johannes Lerch for very useful discussions on the topic. This work was supported by the German Federal Ministry of Education and Research (BMBF) as well as by the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP.

References

[1] S. Arzt and E. Bodden. Reviser: Efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 288–298, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568243. URL `http://doi.acm.org/10.1145/2568225.2568243`.

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, 49(6):259–269, June 2014. ISSN 0362-1340. doi: 10.1145/2666356.2594299. URL `http://doi.acm.org/10.1145/2666356.2594299`.

[3] E. Bodden. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *1st ACM SIGPLAN International Workshop on the State Of the Art in Java Program Analysis*, pages 3–8, 2012. doi: 10.1145/2259051.2259052. URL `http://www.bodden.de/pubs/bodden12inter-procedural.pdf`.

[4] P. Cousot and R. Cousot. *Modular Static Program Analysis*, pages 159–179. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. ISBN 978-3-540-45937-8. doi: 10.1007/3-540-45937-5_13. URL `http://dx.doi.org/10.1007/3-540-45937-5_13`.

[5] M. Eichberg and B. Hermann. A software product line for static analyses: the OPAL framework. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pages 1–6. ACM, 2014.

[6] M. Hanus and F. Skrlac. A modular and generic analysis server system for functional logic programs. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14, pages 181–188, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2619-3. doi: 10.1145/2543728.2543744. URL `http://doi.acm.org/10.1145/2543728.2543744`.

[7] S. Kulkarni, R. Mangal, X. Zhang, and M. Naik. Accelerating program analyses by cross-program training. *SIGPLAN Not.*, 51(10):359–377, Oct. 2016. ISSN 0362-1340. doi: 10.1145/3022671.2984023. URL `http://doi.acm.org/10.1145/3022671.2984023`.

[8] J. Lerch and B. Hermann. Design your analysis: A case study on implementation reusability of data-flow functions. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2015, pages 26–30, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3585-0. doi: 10.1145/2771284.2771289. URL `http://doi.acm.org/10.1145/2771284.2771289`.

[9] J. Lerch, B. Hermann, E. Bodden, and M. Mezini. FlowTwist: Efficient context-sensitive inside-out taint analysis for large codebases. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 98–108, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3056-5. doi: 10.1145/2635868.2635878. URL `http://doi.acm.org/10.1145/2635868.2635878`.

[10] D. Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), Mar. 2014. ISSN 1075-3583. URL `http://dl.acm.org/citation.cfm?id=2600239.2600241`.

[11] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999. ISBN 3540654100.

[12] OSGi Alliance. *Osgi service platform, release 3*. IOS Press, Inc., 2003.

[13] C. W. Probst. *Modular Control Flow Analysis for Libraries*, pages 165–179. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. ISBN 978-3-540-45789-3. doi: 10.1007/3-540-45789-5_14. URL `http://dx.doi.org/10.1007/3-540-45789-5_14`.

[14] D. Ramírez-Deantes, J. Correas, and G. Puebla. *Modular Termination Analysis of Java Bytecode and Its Application to phoneME Core Libraries*, pages 218–236. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-27269-1. doi: 10.1007/978-3-642-27269-1_13. URL `http://dx.doi.org/10.1007/978-3-642-27269-1_13`.

[15] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular software model checking framework. *SIGSOFT Softw. Eng. Notes*, 28(5):267–276, Sept. 2003. ISSN 0163-5948. doi: 10.1145/949952.940107. URL `http://doi.acm.org/10.1145/949952.940107`.

[16] A. Rountev, M. Sharp, and G. Xu. *IDE Dataflow Analysis in the Presence of Large Object-Oriented Libraries*, pages 53–68. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-78791-4. doi: 10.1007/978-3-540-78791-4_4. URL `http://dx.doi.org/10.1007/978-3-540-78791-4_4`.

[17] B. G. Ryder. Incremental data flow analysis. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, pages 167–176, New York, NY, USA, 1983. ACM. ISBN 0-89791-090-7. doi: 10.1145/567067.567084. URL `http://doi.acm.org/10.1145/567067.567084`.

[18] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., 2010.

[19] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: Reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 58:1–58:11, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1614-9. doi: 10.1145/2393596.2393665. URL `http://doi.acm.org/10.1145/2393596.2393665`.